**A. Skateboard REST API**

**1. Technology**:

Used Spring Model-View-Controller (MVC) Framework, Spring Boot Configuration Framework, Apache Maven to build and test, JUnit: an automated unit testing framework, Cucumber: an automated acceptance testing framework, Java 8: writing code, GitHub: commit history for the web service & Travis Continuous Integration (CI): a free continuous integration platform.

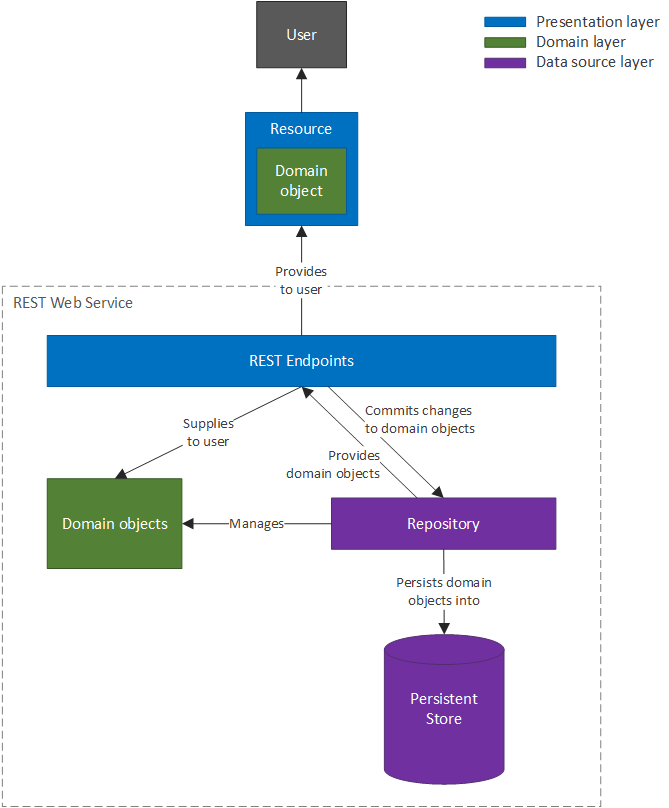
**2. Designing the Web Service**:

Domain Model:

|  |
| --- |
| **Skateboard** |
| * id: Long * owner: String * brand: String * length: BigDecimal * weight: BigDecimal * location: String * status: String * specs: String * created: Date |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | **Tasks** | **HTTP**  **VERB** | **URL** | **Description** |
|  |  | Get skateboards | GET | /board | Obtains the list of all boards currently created in the system. If the list is successfully obtained, the list of existing boards is returned, along with an HTTP 200(OK) status. |
|  |  | Create a skateboard | POST | /board | Creates a new board. This request should carry a request body that includes the data that should be associated with the newly created board. If the board is created, an HTTP 201 (Created) status is returned along with the newly created board in the response body. |
|  |  | Get a skateboard | GET | /board/{id} | Obtains the board associated with the given ID. If no board exists, an HTTP 404 (Not Found) status is returned. If the board can be found, an HTTP 200 status is returned, and the response body contains the information associated with the board. |
|  |  | Update skateboard | PUT | /board/{id} | Updates an existing board. If no board with the given ID can be found, an HTTP 404 status is returned. If a board exists with the given ID and the request body contains valid updates to the board, the board is updated and the updated board is returned in the response body, along with an HTTP 200 status. |
|  |  | Delete skateboard | DELETE | /board/{id} | Deletes a board with the given ID. If no board exists, an HTTP 404 status is returned. If the board exists, it is deleted, and an HTTP 204 (No Content) status is returned. |

The three tiers in this architecture are (1) presentation, (2) domain, and (3) data source (used interchangeably with persistence layer). In this case, the REST endpoints map to the presentation layer, our board domain model maps to the domain layer, and our in-memory database maps to the data source layer.



This design consists four main steps:

1. Implement the domain model
   * Create the Board domain class
2. Implement the data source layer
   * Create an in-memory database
   * Implement the CRUD operations for the Board domain class
3. Implement the presentation layer
   * Create the REST endpoints
   * Create the Board resource
   * Create assembler to construct a Board resource with proper HATEOAS links
4. Pull the application together
   * Create the main method that will run the application

**3. Implementing the Domain Layer**

Obtaining the ID associated with the object:

I do not want the persistence layer to depend on any concrete class in our domain layer for this functionality, so I created an interface to accomplish this task

public interface Identifiable extends org.springframework.hateoas.Identifiable<Long>

{

public void setId(Long id);

}

With the above identity interface completed, now creating Board class:

public class **Board** implements Identifiable {

private Long id;

private String owner;

private String brand;

private BigDecimal weight;

private BigDecimal length;

private String location;

private String status;

private String specs;

private Date created;

@Override

public Long getId()

{

return id;

}

@Override

public void setId(Long id)

{

this.id = id;

}

public String getOwner()

{

return owner;

}

public void setOwner(String owner)

{

this.owner = owner;

}

public String getBrand()

{

return brand;

}

public void setBrand(String brand)

{

this.brand = brand;

}

public BigDecimal getWeight()

{

return weight;

}

public void setWeight(BigDecimal weight)

{

this.weight = weight;

}

public BigDecimal getLength()

{

return length;

}

public void setLength(BigDecimal length)

{

this. length = length;

}

public String getLocation()

{

return location;

}

public void setLocation(String location)

{

this.location = location;

}

public String getStatus()

{

return status;

}

public void setStatus(String status)

{

this.status = status;

}

public String getStatus()

{

return status;

}

public void setStatus(String status)

{

this.status = status;

}

public String getSpecs()

{

return specs;

}

public void setSpecs(String specs)

{

this.specs = specs;

}

public Date getCreated()

{

return created;

}

public void setCreated(String created)

{

this.created = created;

}

}

Since I will be using an in-memory persistence scheme, I cannot rely on a database to provide a new ID for each domain class that I persist. Instead, I will need to create an ID generator that will provide me with unique IDs for each of the Boards we will persist. Our implementation of this generator is as follows:

@Component

@Scope(BeanDefinition.SCOPE\_PROTOTYPE)

public class IdGenerator {

private AtomicLong nextId = new AtomicLong(1);

public long getNextId() {

return nextId.getAndIncrement();

}

}

With ID generator in place, I am now ready to implement our data source. Since much of the in-memory logic is common for all types of objects, I will create an Abstract Base Class (ABC) that contains the core logic for managing our persisted objects (note that Spring uses the nomenclature Repository to

mean a data source, and hence I follow the same convention):

public abstract class **InMemoryRepository**<T extends Identifiable> {

@Autowired

private IdGenerator idGenerator;

private List<T> elements = Collections.synchronizedList(new ArrayList<>());

public T create(T element) {

elements.add(element);

element.setId(idGenerator.getNextId());

return element;

}

public boolean delete(Long id) {

return elements.removeIf(element -> element.getId().equals(id));

}

public List<T> findAll() {

return elements;

}

public Optional<T> findById(Long id) {

return elements.stream().filter(e -> e.getId().equals(id)).findFirst();

}

public int getCount() {

return elements.size();

}

public void clear() {

elements.clear();

}

public boolean update(Long id, T updated) {

if (updated == null) {

return false;

}

else {

Optional<T> element = findById(id);

element.ifPresent(original -> updateIfExists(original, updated));

return element.isPresent();

}

}

protected abstract void updateIfExists(T original, T desired);

}

Since this operation requires information I do not have, I mark it as abstract and require that concrete subclasses provide an implementation of this template method. In this case, I have only one subclass:

@Repository

public class **BoardsRepository** extends InMemoryRepository<Board> {

protected void updateIfExists(Board original, Board updated) {

original.setOwner (updated.getOwner());

original.setBrand(updated.getBrand());

original.setWeight (updated.getWeight());

original.setLocation(updated.getLocation());

original.setStatus(updated.getStatus());

original.setSpecs(updated.getSpecs());

}

}

1. **Implementing the Presentation Layer**

Without the aid of a web application framework, creating a presentation layer would be a daunting task, but after many years, the patterns & conventional designs of RESTful web services have been captured in the Spring Model-View-Controller (MVC) framework. This framework allows us to create RESTful endpoints with much the same ease as I saw during the development of our data source layer, using annotations and helper classes to do most of the heavy lifting for us. Starting with the class that is the most depended on and requires the least dependencies, I will create the BoardResource first:

public class **BoardResource** extends ResourceSupport {

private final long id;

private final String owner;

private final String brand;

private final BigDecimal weight;

private final BigDecimal length;

private final String location;

private final String status;

private final String specs;

private final Date created;

public BoardResource(Board board) {

id = board.getId();

owner = board.getOwner();

brand = board.getBrand();

weight = board.getWeight();

length = board.getLength();

location = board.getLocation();

status = board.getStatus();

specs = board.getSpecs();

created = board.getCreated();

}

@JsonProperty("id")

public Long getResourceId() {

return id;

}

public String getOwner() {

return owner;

}

public String getBrand() {

return brand;

}

public String getWeight() {

return weight;

}

public String getLength() {

return length;

}

public String getLocation() {

return location;

}

public String getStatus() {

return status;

}

public String getSpecs() {

return specs;

}

public String getCreated() {

return created;

}

}

Instead, I can annotate the nested Boardobject with the Jackson @JsonUnwrapped annotation, which removes the nesting when the BoardResource object is serialized. Such an implementation would result in the following definition for the BoardResource class:

public class BoardResource extends ResourceSupport {

@JsonUnwrapped

private final Board board;

public BoardResource(Board board) {

this. board = board;

}

}

Serializing this class would result in our desired **JSON**:

{

"id": 1,

"owner": "Owner1 Name ",

"brand": “Audi”,

"weight": 1979.47,

"length": 59.65

" location": “Detroit”,

"status": “Available”,

"specs": “45x56x25”,

" created": “03/13/2020”

}

With our resource class in place, I need to implement an assembler that will create an BoardResource from an Board domain object:

public abstract class **ResourceAssembler**<DomainType, ResourceType> {

public abstract ResourceType toResource(DomainType domainObject);

public Collection<ResourceType> toResourceCollection(Collection<DomainType> domainObjects) {

return domainObjects.stream().map(o -> toResource(o)).collect(Collectors.toList());

}

}

In implementation of toResourceCollection, we simply map the consumed list of Board objects to BoardResource objects by calling the toResource method on each of the Board objects in the consumed list. We then create a BoardResourceAssembler class that provides an implementation for the toResource method:

@Component

public class BoardResourceAssembler extends ResourceAssembler< Board , BoardResource> {

@Autowired

protected EntityLinks entityLinks;

private static final String UPDATE\_REL = "update";

private static final String DELETE\_REL = "delete";

@Override

public BoardResource toResource(Board  board) {

BoardResource resource = new BoardResource(board);

final Link selfLink = entityLinks.linkToSingleResource(board);

resource.add(selfLink.withSelfRel());

resource.add(selfLink.withRel(UPDATE\_REL));

resource.add(selfLink.withRel(DELETE\_REL));

return resource;

}

}

Given the three links I have created, our resulting BoardResource, when serialized to JSON, looks as follows:

{

"id": 1,

"owner": "Owner1 Name ",

"brand": “Audi”,

"weight": 979.47,

"length": 59.65,

" location": “Detroit”,

"status": “Available”,

"specs": “45x56x25”,

" created": “03/13/2020”

"\_links": {

"self": {

"href": "http://localhost:8080/board/1"

},

"udpate": {

"href": "http://localhost:8080/board/1"

},

"delete": {

"href": "http://localhost:8080/board/1"

}

}

}

In the Spring MVC framework, a REST endpoint is created by implementing a controller class (a class annotated with @Controller or @RestController) and adding methods that correspond to the desired REST endpoints. We will list our controller class first and then explain the meaning of each section of code:

@CrossOrigin(origins = "\*")

@RestController

@ExposesResourceFor(Board.class)

@RequestMapping(value = "/board ", produces = "application/json")

public class BoardController {

@Autowired

private BoardRepository repository;

@Autowired

private BoardResourceAssembler assembler;

@RequestMapping(method = RequestMethod.GET)

public ResponseEntity<Collection< BoardResource>> findAllBoards() {

List<Board> boards = repository.findAll();

return new ResponseEntity<>(assembler.toResourceCollection(boards), HttpStatus.OK);

}

@RequestMapping(method = RequestMethod.POST, consumes = "application/json")

public ResponseEntity<BoardResource> createBoard(@RequestBody Board board) {

Board createdBoard = repository.create(board);

return new ResponseEntity<>(assembler.toResource(createdBoard), HttpStatus.CREATED);

}

@RequestMapping(value = "/{id}", method = RequestMethod.GET)

public ResponseEntity<BoardResource> findBoardById(@PathVariable Long id) {

Optional<Board> board = repository.findById(id);

if (board.isPresent()) {

return new ResponseEntity<>(assembler.toResource(board.get()), HttpStatus.OK);

}

else {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

@RequestMapping(value = "/{id}", method = RequestMethod.DELETE)

public ResponseEntity<Void> deleteBoard(@PathVariable Long id) {

boolean wasDeleted = repository.delete(id);

HttpStatus responseStatus = wasDeleted ? HttpStatus.NO\_CONTENT : HttpStatus.NOT\_FOUND;

return new ResponseEntity<>(responseStatus);

}

@RequestMapping(value = "/{id}", method = RequestMethod.PUT, consumes = "application/json")

public ResponseEntity<BoardResource> updateBoard(@PathVariable Long id, @RequestBody Board updatedBoard) {

boolean wasUpdated = repository.update(id, updatedBoard);

if (wasUpdated) {

return findBoardById(id);

}

else {

return new ResponseEntity<>(HttpStatus.NOT\_FOUND);

}

}

}

For example, if I look at the findAllBoards method,

@RequestMapping(method = RequestMethod.GET)

public ResponseEntity<Collection<BoardResource>> findAllBoards() {

List<Board> boards = repository.findAll();

return new ResponseEntity<>(assembler.toResourceCollection(boards), HttpStatus.OK);

}

1. **Pulling it Together**

With our REST endpoints defined, I am now ready to create the main method that will be executed to start our RESTful web service.

The main method used to start our web service is as follows:

@EnableEntityLinks

@EnableHypermediaSupport(type = HypermediaType.HAL)

@SpringBootApplication

public class Application {

public static void main(String[] args) {

SpringApplication.run(Application.class, args);

}

}

1. **Running and Consuming the Web Service**

Since I am using Maven to manage the dependencies and build the lifecycle of our application, and Spring Boot to configure our application, I can build our project and start the HTTP server using the following command (once Maven has been installed):

**mvn spring-boot:run**

This will host the REST web service on http://localhost:8080. If we look closely at the output, I can see the following statements:

INFO 15204 --- [ main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/board/{id}],methods=[PUT],produces=[application/json]}" onto public org.springframework.http.ResponseEntity<com.dzone.albanoj2.example.rest.resource.BoardResource> com.dzone.albanoj2.example.rest.controller.BoardController.updateBoard(java.lang.Long,com.dzone.albanoj2.example.rest.domain.Board)

INFO 15204 --- [ main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/board],methods=[POST],produces=[application/json]}" onto public org.springframework.http.ResponseEntity<com.dzone.albanoj2.example.rest.resource.BoardResource> com.dzone.albanoj2.example.rest.controller.BoardController.createBoard(com.dzone.albanoj2.example.rest.domain.Board)

INFO 15204 --- [ main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/board/{id}],methods=[GET],produces=[application/json]}" onto public org.springframework.http.ResponseEntity<com.dzone.albanoj2.example.rest.resource.BoardResource> com.dzone.albanoj2.example.rest.controller.BoardController.findBoardById(java.lang.Long)

INFO 15204 --- [ main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/board],methods=[GET],produces=[application/json]}" onto public org.springframework.http.ResponseEntity<java.util.Collection<com.dzone.albanoj2.example.rest.resource.BoardResource>> com.dzone.albanoj2.example.rest.controller.BoardController.findAllBoards()

INFO 15204 --- [ main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/board/{id}],methods=[DELETE]

**B. Project: Boardgroove REST API**

**Option a: used node.js**

First, I need to create a project directory. Next, open the command prompt and navigate to my project directory. Once there, I need to call npm using the below command:

|  |  |
| --- | --- |
|  | npm init |

When I hit enter, Node.js will ask me to enter some details to build the .json file such as:

I will be using **script.js** as an entry point. Next, I will be installing Express.js using the below command:

|  |  |  |
| --- | --- | --- |
|  | **npm i express** |  |
|  |  |  |

Finally, I will be installing a node monitoring package called **nodemon.** It keeps a watch on all the files with any type of extension present in this folder.

**npm i -g nodemon**

**package.json**

|  |  |
| --- | --- |
|  | {  "name": "Boardgroove REST api",  "version": "1.0.0",  "description": "Skateboarders REST API with Node.js",  "main": "script.js",  "scripts": {  "test": "echo "Error: no test specified" && exit 1"  },  "id": 1,  "owner": "Owner 1",  "brand": “Audi”,  "weight": 979.47,  "length": 59.65,  " location": “Detroit”,  "status": “Available”,  "specs": “45x56x25”,  " created": “03/13/2020”  }  }  **script.js** |

**const** express = require('express');

**const** Joi = require('joi'); //used for validation

**const** app = express();

app.use(express.json());

**const** boards = [

{owner: 'Owner 1', id: 1},

{owner: 'Owner 2', id: 2},

{owner: 'Owner 3', id: 3}

]

//READ Request Handlers

//Return all boards

app.get('/', (req, res) => {

res.send('Welcome to Boardgroove REST API with Node.jsl!!');

});

app.get('/api/v1/boards', (req,res)=> {

res.send(boards);

});

//Return a single board

app.get('/api/v1/boards/:id', (req, res) => {

**const** board = boards.find(c => c.id === parseInt(req.params.id));

**if** (!board) res.status(404).send('<h2 style="font-family: Malgun Gothic; color: darkred;">Ooops... Cant find what you are looking for!</h2>');

res.send(board);

});

//CREATE Skateboards

app.post('/api/v1/boards', (req, res)=> {

**const** { error } = validateBoard(req.body);

**if** (error){

res.status(400).send(error.details[0].message)

**return**;

}

**const** board = {

id: boards.length + 1,

owner: req.body.owner,

brand: req.body.brand,

weight: req.body.weight,

length: req.body.length,

location: req.body.location,

status: req.body.status,

photo: req.body.photo,

dimension: req.body.dimension,

specs: req.body.specs,

created: req.body.created;

boards.push(board);

res.send(board);

});

//UPDATE Request Handler

//Update skateboards

app.put('/api/v1/boards/:id', (req, res) => {

**const** board = boards.find(c=> c.id === parseInt(req.params.id));

**if** (!board) res.status(404).send('<h2 style="font-family: Malgun Gothic; color: darkred;">Not Found!! </h2>');

**const** { error } = validateBoard(req.body);

**if** (error){

res.status(400).send(error.details[0].message);

**return**;

}

board.status = req.body.status,

board.specs = req.body.specs;

res.send(board);

});

//DELETE Request Handler

//Delete skateboard

app.delete('/api/v1/boards/:id', (req, res) => {

**const** board = boards.find( c=> c.id === parseInt(req.params.id));

**if**(!board) res.status(404).send('<h2 style="font-family: Malgun Gothic; color: darkred;"> Not Found!! </h2>');

**const** index = boards.indexOf(board);

boards.splice(index,1);

res.send(board);

});

function validateBoard(board) {

**const** schema = {

brand: Joi.string().min(10).required()

};

**return** Joi.validate(board, schema);

}

//PORT ENVIRONMENT VARIABLE

**const** port = process.env.PORT || 8080;

app.listen(port, () => console.log(`Listening on port ${port}..`));

Now, the next step is to check whether the handlers are working properly or not. For that, I will use a Chrome extension called Postman. click on ‘Add to Chrome’. Once installed Postman, open it and start testing your application. So, let’s start off by testing our GET method. Now, in order to do that I need to select GET from the drop-down list, type in the defined URI and hit send. If code is working fine, then I will see the list of all the boards which I have added manually in the code.

Now, let’s try adding a new board to the inventory list. For that, select ‘POST’ from the drop-down list and type in the defined URI for the POST method. Now, click on ‘Body’, select ‘raw’ and move on to select ‘JSON’ from the drop-down list as depicted. Now, in the text area, type in the of board and hit send.

If the POST method is working fine, the response body will contain the board details along with the board id. Now, let’s try to update the board owner. Currently, my board owner is “Owner1” which I will be updating to “Owner2”. So, to update the data, need to first select ‘PUT’ from the drop-down table and enter the PUT request’s URI along with the board id you wish to update. Next in the ‘Body’, type in the new board owner and hit enter.

Finally, let’s send a ‘DELETE’ request to delete an existing record. For that select DELETE from the drop-down list and type in the URI of the delete request handler along with the board details, I want to remove and hit enter. If transaction is successful, I will see the complete details of the entry you have removed in the response body.

**Option b: Used swagger**

**SwaggerConfig.java:**

package guru.springframework.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.RequestHandlerSelectors;

import springfox.documentation.service.ApiInfo;

import springfox.documentation.service.Contact;

import springfox.documentation.spi.DocumentationType;

import springfox.documentation.spring.web.plugins.Docket;

import springfox.documentation.swagger2.annotations.EnableSwagger2;

import static springfox.documentation.builders.PathSelectors.regex;

@Configuration

@EnableSwagger2

public class SwaggerConfig {

@Bean

public Docket boardApi() {

return new Docket(DocumentationType.SWAGGER\_2)

.select()

.apis(RequestHandlerSelectors.basePackage("guru.springframework.controllers"))

.paths(regex("/board.\*"))

.build()

.apiInfo(metaData());

}

private ApiInfo metaData() {

ApiInfo apiInfo = new ApiInfo(

"Boardgroove REST API",

" Boardgroove REST API for skateboard",

"Apache License Version 2.0",

"https://www.apache.org/licenses/LICENSE-2.0");

return apiInfo;

}

}

In the SwaggerConfig class, added a metaData() method that returns and ApiInfo object initialized with information about our API. Line 23 initializes the Docket with the new information.

The Swagger 2-generated documentation now looks similar to this:

**Swagger 2 Annotations for REST Endpoints**

At this point, if you click the board controller link, Swagger UI will display the documentation of the operation endpoints, like this:

I can use the @Api annotation on our BoardController class to describe this API.

@RestController @RequestMapping("/board") @Api(value="onlinestore", description="Operations pertaining to boards in Online Store") public class BoardController { . . . . }

The Swagger UI-generated documentation will reflect the description and now looks like this:  
For each of our operation endpoints, I can use the @ApiOperation annotation to describe the endpoint and its response type, like this:

@ApiOperation(value = "View a list of available boards", response = Iterable.class)

@RequestMapping(value = "/list", method= RequestMethod.GET,produces = "application/json")

public Iterable list(Model model){

Iterable boardList = boardService.listAllBoards();

return boardList;

}

Swagger 2 also allows overriding the default response messages of HTTP methods. I can use the @ApiResponse annotation to document other responses, in addition to the regular HTTP 200 OK, like this.

@ApiOperation(value = "View a list of available boards", response = Iterable.class)

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved list"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found")

}

)

@RequestMapping(value = "/list", method= RequestMethod.GET, produces = "application/json")

public Iterable list(Model model){

Iterable boardList = boardService.listAllBoards();

return boardList;

}

One undocumented thing that took quite some of my time was related to the value of Response Content Type. Swagger 2 generated \*/\*, while I was expecting application/json for Response Content Type. It was only after updating the @RequestMapping annotation, which produces = "application/json", that the desired value got generated. The annotated BoardController is below.

**BoardController.java:**

package guru.springframework.controllers;

import guru.springframework.domain.Board;

import guru.springframework.services.BoardService;

import io.swagger.annotations.Api;

import io.swagger.annotations.ApiOperation;

import io.swagger.annotations.ApiResponse;

import io.swagger.annotations.ApiResponses;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.ui.Model;

import org.springframework.web.bind.annotation.\*;

@RestController

@RequestMapping("/board")

@Api(value="Board Groove", description="Operations pertaining to boards in Board Groove")

public class BoardController {

private BoardService boardService;

@Autowired

public void setBoardService(BoardService boardService) {

this.boardService = boardService;

}

@ApiOperation(value = "View a list of available boards",response = Iterable.class)

@ApiResponses(value = {

@ApiResponse(code = 200, message = "Successfully retrieved list"),

@ApiResponse(code = 401, message = "You are not authorized to view the resource"),

@ApiResponse(code = 403, message = "Accessing the resource you were trying to reach is forbidden"),

@ApiResponse(code = 404, message = "The resource you were trying to reach is not found")

}

)

@RequestMapping(value = "/list", method= RequestMethod.GET, produces = "application/json")

public Iterable<Board> list(Model model){

Iterable<Board> boardList = boardService.listAllBoards();

return boardList;

}

@ApiOperation(value = "Search a board with an ID",response = Board.class)

@RequestMapping(value = "/show/{id}", method= RequestMethod.GET, produces = "application/json")

public Board showBoard(@PathVariable Integer id, Model model){

Board board = boardService.getBoardById(id);

return board;

}

@ApiOperation(value = "**Create a board**")

@RequestMapping(value = "/add", method = RequestMethod.POST, produces = "application/json")

public ResponseEntity saveBoard(@RequestBody Board board){

boardService.saveBoard(board);

return new ResponseEntity("Board saved successfully", HttpStatus.OK);

}

@ApiOperation(value = "**Update a board**")

@RequestMapping(value = "/update/{id}", method = RequestMethod.PUT, produces = "application/json")

public ResponseEntity updateBoard(@PathVariable Integer id, @RequestBody Board board){

Board storedBoard = boardService.getBoardById(id);

storedBoard.setStatus(board.getStatus());

storedBoard.setSpecs(board.getSpecs ());

boardService.saveBoard(storedBoard);

return new ResponseEntity("Board updated successfully", HttpStatus.OK);

}

@ApiOperation(value = "**Delete a board**")

@RequestMapping(value="/delete/{id}", method = RequestMethod.DELETE, produces = "application/json")

public ResponseEntity delete(@PathVariable Integer id){

boardService.deleteBoard(id);

return new ResponseEntity("Board deleted successfully", HttpStatus.OK);

}

}

**Summary**

Besides REST API documentation and presentation with Swagger Core and Swagger UI, Swagger 2 has a whole lot of other uses beyond the scope of this. One of my favorites is [Swagger Editor](http://swagger.io/swagger-editor/), a tool to design new APIs or edit existing ones. The editor visually renders your Swagger definition and provides real-time error-feedback. Another one is [Swagger Codegen](http://swagger.io/swagger-codegen/), a code generation framework for building Client SDKs, servers, and documentation from Swagger definitions.

Swagger 2 also supports Swagger definition through [JSON](https://dzone.com/articles/json-is-here) and [YAML](https://dzone.com/articles/in-defense-of-yaml) files. It is something I did try if I want to avoid implementation-specific code in your codebase by externalizing them in JSON and YAML files.